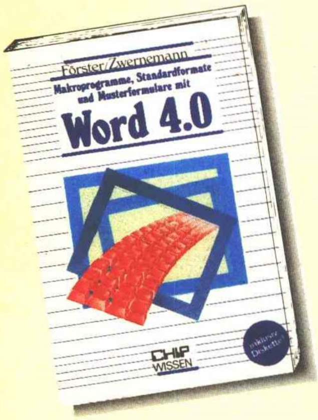
CHIP PLUS

EINE BEILAGE
IN CHIP NR. 3
MÄRZ 1989

ADA

CHIP WISSEN

Die kompetente Buchreihe rund um den PC



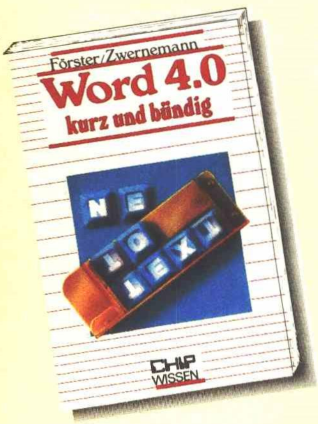
Hans-Peter Förster/Martin Zwernemann:

Makroprogramme, Standardformate und Musterformulare mit Word 4.0

144 Seiten, 42 Bilder, Hardcover
48,– DM (mit 5,25"-Diskette)
ISBN 3-8023-**0246**-X

Jeder der mit Word 4.0 Standardformulare ausfüllt, Serientexte schreibt und Etiketten bedruckt oder eigene Makroprogramme und Druckformatvorlagen erstellen möchte, braucht dieses Buch mit Diskette.

Auf der Diskette befinden sich nahezu alle Etikettenformate, Formate für Endlospapiere und Standardformulare.
Die Makroprogrammierung, das Arbeiten mit Druckformaten und Serientexten wird Schritt für Schritt anhand von Praxis-Beispielen erklärt.



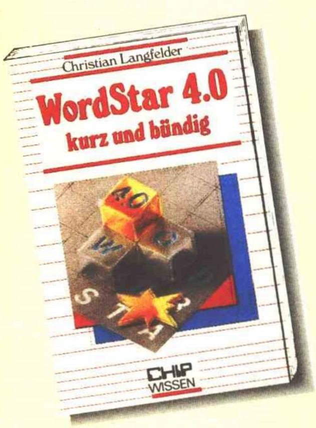
Hans-Peter Förster/Martin Zwernemann:

Word 4.0 kurz und bündig

256 Seiten, 67 Bilder, Hardcover
43,– DM/ISBN 3-8023-**0215**-X

Dieses Arbeitsbuch richtet sich an interessierte Einsteiger sowie an Word-4.0-Anwender, die schnell den Umgang mit dem komfortablen Textverarbeitungsprogramm erlernen wollen. Es ist so aufbereitet, daß es auch als Begleitmaterial für Word-4.0-Schulungen geeignet ist. Aber auch zum Selbststudium ist es in Ergänzung zum Word-4.0-Handbuch ein Hilfsmittel, das viele Tips und Tricks verrät.

* Wichtige Grundfunktionen
* Texte erstellen, korrigieren, speichern
* Texte laden, korrigieren, gestalten
* Texte direkt/indirekt formatieren
* Ausdruck
* Textbausteine
* Rechtschreibprogramm u.a.

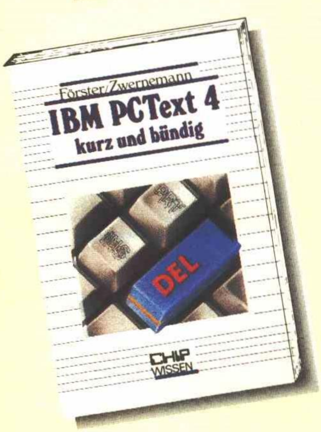


Christian Langfelder:

WordStar 4.0 kurz und bündig

184 Seiten, 46 Bilder, Hardcover
35,– DM/ISBN 3-8023-**0219**-2

Dieses Buch verfolgt einerseits das bewährte Konzept seines Vorgängers „WordStar kurz und bündig" und macht den Leser andererseits mit den neuen Eigenschaften und Möglichkeiten von WordStar 4.0 bzw. den Zusatzprogrammen von WordStar 4.0 Extra vertraut. Es ist somit für Einsteiger in der Textverarbeitung als auch für Umsteiger auf die neue Version bestens geeignet.



Hans-Peter Förster/Martin Zwernemann:

IBM PCText 4 kurz und bündig

200 Seiten, 47 Bilder, Hardcover
38,– DM/ISBN 3-8023-**0247**-8

Ziel dieses Buches ist es, dem Anwender des IBM Textverarbeitungsprogrammes PCText 4 einen Ratgeber zur Seite zu stellen, der alle Funktionen und Befehle knapp aber präzise vorstellt.

Immer wieder wird der Leser beim Durcharbeiten des Buches zur direkten Eingabe der notwendigen Tasten aufgefordert. Die Knappheit des Textes und die Fülle der Informationen machen das Buch zu einem wichtigen Nachschlagewerk bei der täglichen Arbeit und damit zu einer praktischen Ergänzung zum IBM-Handbuch.

Haben Sie schon den neuen „CHIP-Katalog"
Bestellen Sie gleich!

VOGEL Buchverlag Würzburg
Postfach 6740 · 8700 Würzburg 1

Ada und die Software-Krise

Ulrich Parthier

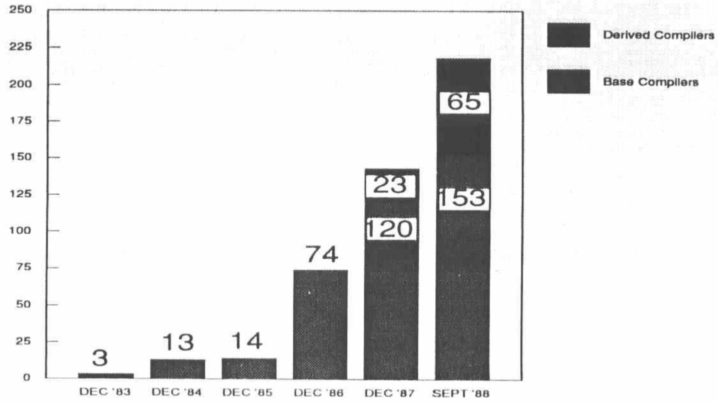
Wenn man über die Programmiersprache Ada spricht, dann fällt mit schöner Regelmäßigkeit in diesem Zusammenhang auch der Begriff der Software-Krise. Kaum eine andere Programmiersprache scheint nach den Aussagen vieler Experten geeigneter, dieses Problem besser in den Griff zu bekommen als Ada.

Ada und ihre Entstehung sind eng mit dem amerikanischen Verteidigungsministerium, dem Department of Defense (DoD), verbunden. Das DoD hatte seit den siebziger Jahren angesichts des steigenden Bedarfs an Software, der Kostenlawine bei ihrer Entwicklung und der Ansprüche an sie (Fehlerfreiheit, Wartbarkeit, Echtzeit-Anforderungen) überlegt, wie man diese Probleme am besten lösen könne. Trotz der Vielfalt der bestehenden Programmiersprachen konnte keine das DoD zufriedenstellen. Aus einer Ausschreibung des Ministeriums ging schließlich Ada als Siegerin hervor.

Die Sprache selbst hat eigentlich wenig Revolutionäres zu bieten – das bestätigen auch ihre Schöpfer. Doch Ada ist eben viel mehr als eine Programmiersprache. Ihr wesentliches Plus verdankt sie den Design-Möglichkeiten, ihrer Methodik. Software kann mit Ada schneller und kostengünstiger produziert werden. Der Qualitätsstandard von Ada-Programmen liegt von vornherein erheblich höher als bei anderen Hochsprachen – und sie sind leichter zu warten.

Ein Blick auf diverse Untersuchungen (etwa des Carnegie Mellon Software Engineering Institute oder des US-Government Accounting Office) zeigt Erschreckendes: Rund drei Viertel aller in Angriff genommenen Software-Projekte werden entweder gar nicht oder nur mit groben Mängeln abgeschlossen und nicht in Betrieb genommen, ganze zwei Prozent sind ohne jegliche Änderung lauffähig.

An Möglichkeiten, die Software-Lücke zu schließen, fehlt es ja nicht. So stehen heute im Vergleich zu früher Tools, Entwicklungsumgebungen, Codegeneratoren etc. als Hilfsmittel zur Verfügung. Und an der fehlenden Programmiersprache, wie gesagt, mangelt es ebenfalls nicht.



Anzahl der validierten Compiler (1983 bis 1988)

Ada macht ernst mit dem Software-Engineering

Michael Tonndorf



Der Autor Michael Tonndorf, 35, ist Diplom-Informatiker und Mitarbeiter der IABG in Ottobrunn bei München. Dort führt er seit drei Jahren Validierungen von Ada-Compilern durch und beschäftigt sich mit Software-Entwicklungsumgebungen für Realzeitanwendungen mit Ada.

Als Werkzeug für das professionelle Software-Engineering wurde die Computer-Sprache Ada geschaffen. Sie unterstützt insbesondere die Entwicklung und Pflege großer Software-Systeme und bietet zu diesem Zweck moderne, leistungsfähige Mittel an.

Eine Programmiersprache und ihr Compiler spiegeln verräterisch wider, wie der Mensch die Fähigkeiten eines Computers auszunutzen gedenkt. Eine einzige Sprache für alle denkbaren Computer-Anwendungen ist jedenfalls eine unzeitgemäße Forderung. Ada ist zwar eine universelle Programmiersprache im Sinne der Informatik. Es wurde aber nie behauptet, Ada vereinige alle bekannten und wünschenswerten Eigenschaften einer Programmiersprache in sich.

Welches sind denn aber die Eigenschaften, die Ada von anderen Programmiersprachen unterscheiden und die Bedeutung von Ada ausmachen?

Zweifellos ist die Normierung die wichtigste Anforderung – noch vor allen fachlichen Konzepten, die Ada auf den Weg gegeben worden sind. Oberstes Ziel war es, einen verbindlichen Sprachstandard zu schaffen und in der Praxis durchzusetzen.

Diesem Entschluß gingen leidvolle Erfahrungen mit Compilern oder Assemblern einer großen Zahl unterschiedlichster Computer-Sprachen voraus. Nicht nur, daß es für manche Sprachen mehrere voneinander abweichende Definitionen gab. Auch die Umsetzung durch die verschiedenen Compiler hatte meist ihre Besonderheiten, indem etwa Gegebenheiten einer speziellen Maschine besonders gut ausgenutzt und dafür gewisse Sprachbestandteile großzügig vernachlässigt wurden. Das Resultat war in jedem Fall, daß Programme trotz aller Normen nur mit großem Aufwand von einer zur anderen Maschine transportiert werden konnten und daß Programmierer sich in einer ungewohnten Umgebung wieder von vorn einarbeiten mußten.

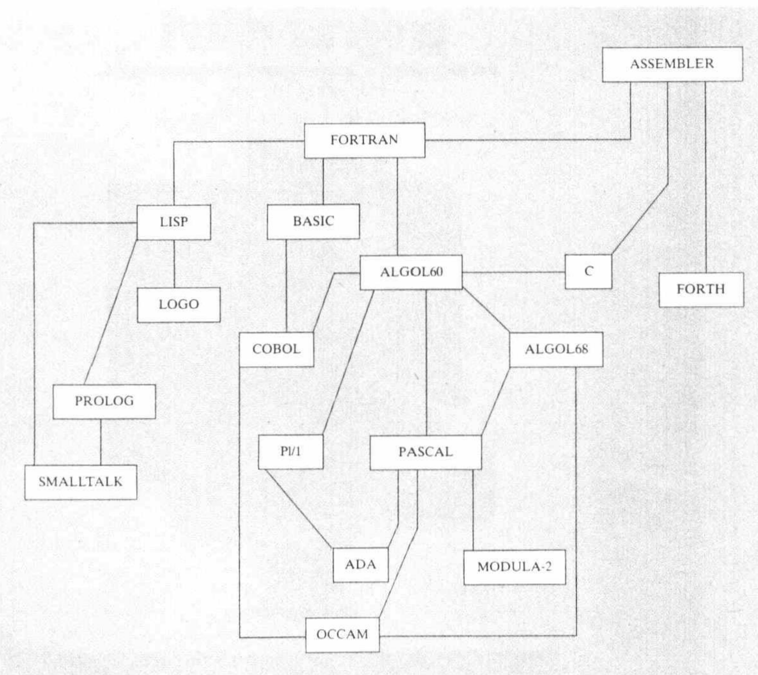
Die Normierung von Ada (1983 US-Standard, 1987 ISO-Standard) bedeutet nun nicht, daß ein Programm in dieser Sprache einer Umgebung ohne jegliche Anpassung auf einer anderen Umgebung zum Laufen gebracht werden kann. Es bedeutet aber, daß man dem Benutzer-Manual des Compilers leicht ansehen kann, worin die Möglichkeiten und Restriktionen dieser Sprache bestehen. Damit kann man auch leicht erkennen, welche Programmteile implementierungsabhängig sind und bei einer Portierung geändert werden müssen.

Die Sprachdefinition von Ada verlangt nämlich auch, daß alle maschinenabhängigen Größen, die für den Benutzer eine Rolle spielen (beispielsweise größter Integer-Wert oder die kleinste Auflösung bei Float-Zahlen), nach einem festen Schema im Manual aufgeführt werden. Erklärtes Ziel bei der Definition von Ada war es, Erkenntnisse aus einem Jahrzehnt Software-Engineering in die Sprache einzubeziehen. Damit ist klar, daß Ada speziell die Entwicklung großer Software-Systeme unterstützen soll. Klar ist aber auch, daß sie Konzepte und Mechanismen enthält, die bei der getrennten Übersetzung von Programmeinheiten nützlich sind. Schließlich hat sich die Disziplin Software-Engineering zuerst aus den Schwierigkeiten entwickelt, die beim „Programmieren im Großen" entstanden waren.

Getrennte Übersetzung bedeutet nichts anderes, als daß Teile eines späteren lauffähigen Programms in unabhängigen Compiler-Durchläufen erzeugt werden. Da getrennt übersetzte Programmeinheiten im laufenden Programm im allgemeinen Daten austauschen, muß bei der getrennten Übersetzung ein entsprechender Austauschmechanismus bereitgestellt werden. Und genau dieser Austausch war die Schwachstelle bei den herkömmlichen Programmiersprachen.

Wenn eine Sprache überhaupt die getrennte Übersetzung unterstützt, dann wird in der Regel nicht überprüft, ob die über Modulgrenzen hinweg benutzten

Größen auch typgerecht verwendet werden (von der bekannten Ausnahme Modula-2 abgesehen). Wenn etwa in einem Modul M1 die Variable tx als ganzzahlige Größe vereinbart wurde, ein Modul M2 die Variablen von Modul M1 benutzen darf und in M2 an tx eine Zeichenkette zugewiesen wird, dann soll schon zur Überset-



Genealogie der Programmiersprachen

zungszeit ein Fehler festgestellt werden. Das setzt natürlich voraus, daß bei der Übersetzung von M2 der Typ von tx an einer geeigneten Stelle einzusehen ist.

Dies kann nur effizient geschehen, wenn dabei M1 nicht auch noch einmal übersetzt werden muß, sonst hätte die getrennte Übersetzung ja keinen besonderen Vorteil mehr. Die Schnittstellenkonsistenz, die in diesem Beispiel noch als Trivialität erscheint, ist in einem großen Projekt mit mehreren Programmier-Teams und vielen Übersetzungseinheiten längst keine Selbstverständlichkeit mehr.

Eine Lösung dieses Problems besteht in der Einrichtung einer Programmbibliothek, in der alle relevanten Informationen über Übersetzungseinheiten aufbewahrt werden. Die Bibliothek muß vom Compiler selbst verwaltet werden, damit kein gewitzter Benutzer die Gelegenheit hat, an der Bibliotheksverwaltung vorbei zu übersetzen.

An diesem Beispiel ist bereits zu sehen, wie eine Ada-Übersetzungseinheit die Größen einer anderen Bibliothekseinheit benutzen kann: Vor die Deklaration von M2 muß lediglich die Kontextklausel „with M1;" gesetzt werden. Damit sind die sichtbaren Größen (Objekte, Typen, Unterprogramme) von M1 auch in M2 benutzbar. Durch diese „with"-Klausel wird eine „benutzt"-Relation auf den Elementen einer Ada-Bibliothek definiert. Diese Relation impliziert eine Reihenfolge von Nachübersetzungen, die fällig werden, wenn eine bereits in der Bibliothek befindliche Einheit neu übersetzt wird. Durch diese Übersetzung erhalten nämlich alle von dieser Einheit abhängigen Einheiten den Status „ungültig" und können zum Binden nicht mehr herangezogen werden.

Dies ist aber noch längst nicht alles, was Ada an Unterstützung für die getrennte Übersetzung zu bieten hat. Die grundlegende Konstruktion, die Ada für diesen Zweck bereithält, ist die Trennung einer Programmeinheit in die Bestandteile Spezifikation und Rumpf. Die Spezifikation ist dabei die Ansammlung aller Größen der Programmeinheit, die sie der Außenwelt zur Verfügung stellt. Der Rumpf hingegen enthält die eigentliche Implementierung, das heißt die algorithmischen Elemente und auch etwaige lokale Deklarationen, die nicht für die Außenwelt bestimmt sind.

Als weiteres Sprachmittel zur Kontrolle der Benutzung von Größen aus anderen Programmeinheiten gibt es in Ada die „privaten Typen" und die „streng privaten Typen". Objekte dieser Art stehen der Außenwelt nur in eingeschränkter Form zur Verfügung. Dieses Verbergen von Datentypen ist auch unter den Begriffen Datenkapselung und Datenabstraktion bekannt. Verstöße gegen diese Typregeln werden bereits zur Übersetzungszeit erkannt und als Fehler gemeldet.

Des weiteren können in Ada neben Spezifikation und Rumpf Untereinheiten von Programmeinheiten separat übersetzt werden. Als Untereinheiten sind Rümpfe von Paketen, Prozeduren oder Prozessen möglich, die statt an ihrer regulären Position im Programmtext in einem separaten Compiler-Lauf übersetzt werden. Das heißt, man kann Teile einer größeren Einheit „auslagern" und einzeln übersetzen.

Dieser Vorgang kann geschachtelt werden, so daß eine Top-down-Programmentwicklung möglich wird. Die Untereinheiten sind dabei fest an die Bibliothekseinheit gekoppelt, zu der sie gehören, und können von anderen Bibliothekseinheiten nicht benutzt werden. Für die „Wiederver-

wendung" von Software eignet sich das Konzept der separaten Untereinheiten also nicht.

Nun läßt andererseits eine Ada-Programmeinheit dem wiederverwendungswilligen Programmierer nicht allzu viele Freiheiten, sondern gerade das, was man vom Parameterübergabe-Mechanismus als formale und aktuelle Parameter kennt. Zusammen mit dem strengen Typ-Konzept von Ada wäre dies keine Lösung des Wiederverwendungsproblems. So bietet Ada auch ein mächtigeres Verfahren an, das die Parametrierung nicht nur von Objekten, sondern auch von Typen und Unterprogrammen erlaubt.

In Ada bilden „generische Programmeinheiten" die Basis eines Wiederverwendungskonzepts. Eine generische Programmeinheit ist eine Programmschablone, die für sich nicht ablauffähig ist. Erst durch eine Ausprägung (engl.: instantiation) wird diese Schablone zu einer Programmeinheit, die wie jede andere Ada-Programmeinheit Teil eines lauffähigen Ada-Programms werden kann.

Beispiel für das generische Konzept

Berühmtestes Beispiel für die Anwendung des generischen Konzepts ist wohl die Warteschlange. Die typischen Operationen der Warteschlangen-Verwaltung „kreieren", „löschen", „anhängen", „wegnehmen", „ist – voll", „ist – leer" können verwirklicht werden, ohne daß der Typ der Elemente der Warteschlange bekannt ist. Dies ist bei allen herkömmlichen Programmiersprachen unmöglich, es sei denn, man verwendet krumme Tricks, etwa typungebundene Zeiger.

Die Ausprägung der Warteschlangen-Verwaltung für einen ganz bestimmten Typ sieht dann ähnlich wie ein Unterprogrammaufruf aus. Dieses Konzept wird in Ada aber auch dafür benutzt, um Funktionen als Parameter zu übergeben, eine schon seit Algol 60 bekannte Technik.

Einer der Leitgedanken bei der Definition von Ada war es, so viele Fehler und Ungenauigkeiten wie möglich schon zur Übersetzungszeit abzufangen. Dieser Anspruch bedingt, daß man bereits zur Übersetzungszeit jedes Objekt einem Datentyp zuordnen kann. Der vielstrapazierte Begriff „Objekt" ist deshalb bei Ada auch ganz lapidar definiert als „Größe, die einen Wert eines vorgegeben Typs besitzt".

Dieses Verständnis von Datentypen wird deshalb allgemein auch als das „starke Typ-Konzept" bezeichnet. Daß damit die Ziele des Software-Engineering unterstützt werden, liegt auf der Hand: Hier geht es vor allem um die Erkenntnis, daß es insgesamt viel billiger und zeitsparender ist, mögliche Fehler in frühen Phasen der Software-Entwicklung abzufangen und gar nicht erst in laufende Programme einnisten zu lassen.

Die Väter von Ada hatten beim Sprachentwurf vor allem die Entwicklung von Software für systemintegrierte Computer („Embedded Systems") im Auge. Deshalb werden die Einrichtung von Prozessen und die Kommunikation zwischen ihnen von der Sprache besonders unterstützt. Ada bietet das Rendezvous als Sprachmittel an, das an anderer Stelle in diesem Heft besprochen wird. Andere Systeme unterstützen andere Mechanismen für die Synchronisation von Prozessen (z.B. die Koroutinen von Modula-2 oder das Semaphor-Konzept, das unter anderem im Betriebssystem OS/2 eingesetzt wird).

Diese Methoden sind in gewisser Weise äquivalent, wobei in Ada der Nachrichten-Übergabemechanismus wohl am elegantesten gelöst ist. Andererseits hat Ada im Bereich der Ein-/Ausgabe nichts vorzuweisen, was es von den klassischen Konkurrenten unterscheiden würde. Gleichwohl sind beispielsweise GKS- und SQL-Schnittstellen längst vorhanden, so daß mit Ada als Implementierungssprache auf Lösungen in anderen Anwendungsgebieten zurückgegriffen werden kann.

Insgesamt ist der besondere Vorteil der Programmiersprache Ada in der Summe ihrer Möglichkeiten zu sehen. Diese ergeben sich aus

- den klassischen Konzepten der sequentiellen und prozeduralen Ablaufsteuerung,
- den hochaggregierten Sprachmitteln der Prozeßimplementierung und Prozeßkommunikation einschließlich der Möglichkeit der maschinennahen Programmierung,
- in Verbindung mit der integrierten Unterstützung von Software-Engineering-Prinzipien.

Wer sich vom Umfang des Ada-Sprachmanuals überwältigt sieht, der sollte nicht vergessen, was der Grundgedanke des Sprachentwurfs war. Ada ist mehr als eine Sprache zum Abfassen von Algorithmen.

Die Validierung von Ada-Compilern

Michael Tonndorf

In dem Beitrag über den Software-Engineering-Aspekt von Ada wurde darauf hingewiesen, daß eine einheitliche und verbindliche Sprachnorm für Ada eines der Hauptanliegen der „Sponsoren" und Entwickler der Sprache war. Schließlich haben die negativen Erfahrungen mit dem Sprachen- und Dialekt-Wirrwarr in den siebziger Jahren den Impuls zur Entwicklung von Ada geliefert.

Verlangt man die Normgerechtheit einer Sprache, dann muß man dies auch für einen Compiler fordern. Und die Normgerechtheit für einen Ada-Compiler zu überprüfen, ist keine einfache Aufgabe. Das US-Verteidigungsministerium (DoD, Department of Defense) hat deshalb einige Institutionen und Gremien ins Leben gerufen, die die Pflege der Sprache und die Überwachung der Sprachnorm zum Ziel haben. Zuerst wäre das Ada Joint Program Office (AJPO) zu nennen, das als zentrale Koordinierungsstelle im DoD und „Projektbüro" für Ada fungiert. Alle Informationen und Entscheidungen bezüglich Ada laufen hier ab. Dem AJPO angegliedert ist das Ada Information Clearing House (AdaIC), die Informationsbörse für alle Ada-Fragen. Eine der wichtigsten Veröffentlichungen des AdaIC ist sicherlich die Liste der validierten Compiler, die monatlich aktualisiert wird. Im hier betrachteten Zusammenhang ist als Ada-Gremium nur noch das Ada Validation Office (AVO) von Interesse. Dem AVO obliegt es, für die Validierung von Ada Compilern einheitliche Richtlinien zu erarbeiten. Bevor wir uns also weiter mit organisatorischen Fragen zum Thema Validierung auseinandersetzen, müssen wir an dieser Stelle näher erläutern, was eine Validierung ist, aber auch, was sie nicht ist.

Die Validierungsfrage

Ziel einer Validierung ist es, glaubhaft zu machen, daß ein Compiler für Ada

- Programme, die nach den Sprachregeln abgefaßt sind, in äquivalente Programme der Maschinensprache übersetzt,
- Programme, die Fehler enthalten, nicht in die Maschinensprache übersetzt.

Dies klingt verhältnismäßig schlicht, hat jedoch einige weitreichende Konsequenzen. So stellt sich zuerst die Frage nach einer Testsystematik. Wie soll denn ein Verfahren aussehen, das für die genannten Ziele mit einem beliebigen Ada-Compiler praktikabel ist? Hier ist stets ein Kompromiß zwischen dem Aufwand (also der Kosten) und der Genauigkeit des Ergebnisses einzugehen.

Die bei Ada gewählte Systematik orientiert sich am Aufbau des Ada-Sprachmanuals (Reference Manual for the Ada Programming Language, LRM). In dessen Kapitel und Abschnitten werden alle Sprachbestandteile nach einem einheitlichen Schema definiert. Und so hat jedes Testprogramm ein ganz bestimmtes Detail der Sprachdefinition zum Thema. Dieses Testziel (test objective) steht als Kommentar am Anfang eines jeden Testprogramms. Im „Endausbau" soll die Sammlung der Testprogramme (Ada Compiler Validation Capability ACVC) das ganze Sprachmanual abdecken, inklusive des schwierigen Kapitels 13 (Representation Clauses and Implementation-Dependend Features). Die vollständige Abdeckung ist für die übernächste Version der Testprogramme, ACVC 1.12, angestrebt, zur Zeit sind es zirka 85 – 90%. Des weiteren unterliegen die Tests einer Einteilung in sechs Testklassen.

Klasse	Bedeutung
A	Ausführbare Programme ohne Prüfung des Laufzeitverhaltens des erzeugten Codes,
B	Programme, die Fehlermeldungen zur Übersetzungszeit erwarten lassen,
C	ausführbare Programme mit Prüfung des Laufzeitverhaltens,
D	ausführbare Programme mit Kapazitätsprüfungen,
E	ausführbare Programme, die Aufschluß über implementierungsabhängige Details geben,
L	Tests, die aus mehreren Übersetzungseinheiten bestehen, und die zur Bindezeit Fehler produzieren sollen.

Schließlich wird für jeden Test noch unterschieden, ob er obligatorisch und universell anwendbar ist, ob er implementierungsspezifische Parameter benötigt, oder ob er zur Kategorie der Tests gehört, die möglicherweise nicht anwendbar für eine spezielle Implementierung sind (dependency-Test). Zu der letztgenannten Kategorie gehören zum Beispiel Tests, die die Korrektheit der Implementierung von Sprachbestandteilen überprüfen, die das ARM nicht verpflichtend für jeden Compiler vorschreibt, beispielsweise das Pragma inline.

Als Beispiel für einen ausführbaren Test ist der Test C96005D.ADA angegeben. Der Testname ist im allgemeinen identisch mit dem Dateinamen der Quelle des Tests und ist ein Hinweis darauf, daß der Test der Klasse C angehört, sich auf den Abschnitt 9.6 des ARM bezieht (Delay Statements, Duration, and Time) und er ein obligatorischer Test für alle Ada-Compiler ist. Er wird zusammen mit dem report-package übersetzt, welches Routinen zur Unterstützung des Testablaufs enthält. Der Test ist dann erfolgreich, wenn er keinen der beiden FAILED-Aufrufe, sondern die RESULT-Anweisung am Ende des Programmes erreicht. In diesem Fall wird eine PASSED-Meldung ausgegeben. Alle ausführbaren Tests sind nach diesem Schema aufgebaut. Lediglich für die Klasse-B- und die Klasse-L-Tests gelten andere Auswertungsregeln. Dort ist das Übersetzungs- beziehungsweise Binde-Protokoll dahingehend zu überprüfen, ob alle „eingebauten" Fehler vom Compiler beziehungsweise vom Binder auch tatsächlich gemeldet wurden. Und nun kommt die entscheidende Hürde: Verhält sich der Compiler auch nur bei einem Test nicht erwartungsgemäß, dann ist die Validierung gescheitert! Und dies bei einem Umfang der Test-Suite von derzeit 3 717 Tests, verteilt auf 4 180 Dateien. An dieser Stelle wird auch klar, was eine Validierung nicht ist: sie ist kein Beweis für die Korrektheit eines Compilers und sie ist auch kein Nachweis der Leistungsfähigkeit eines Compilers. Gerade auf den letzten Aspekt, die Bewertung der Güte eines Compilers, ist zur Zeit ein großes Augenmerk der Fachwelt gerichtet. Hierzu sind in den USA, Großbritannien und Deutschland bereits andere Verfahren in Entwicklung (die Evaluation), die aber noch nicht in einem ähnlich einheitlichen Verfahren zur Verfügung stehen wie die Validierung.

Bei einem validierten Ada-Compiler kann man zunächst einmal davon ausgehen, daß die Sprache wirklich so implementiert ist, wie es die Sprachdefinition verlangt. Und das ist mehr, als man von allen anderen Programmiersprachen bisher gewohnt ist.

Der zeitliche Ablauf

Wie läuft nun eine Validierung tatsächlich ab? Zur Abnahme der Validierung hat das AJPO derzeit fünf Organisationen berechtigt (Ada Validation Facilities, AVF's). Neben zwei Stellen in den USA, je einer in Großbritannien und Frankreich führt die Industrieanlagen-Betriebsgesellschaft (IABG) in Ottobrunn in eigener Verantwortung Validierungen von Ada-Compilern durch. So hat die IABG im Gültigkeitszeitraum von ACVC 1. 9. 1987/1988 für 12 Compiler bei sechs verschiedenen Herstellern die Validierung abgenommen. Resultat einer Validierung ist ein Zertifikat, ein Ergebnisbericht (Validation

Der Test C96005D.ADA

```
-- C96005D.ADA

-- CHECK THE CORRECTNESS OF THE ADDITION AND SUBTRACTION FUNCTIONS IN
-- THE PREDEFINED PACKAGE CALENDAR, AND APPROPRIATE EXCEPTION HANDLING.
-- SPECIFICALLY,
--   (D) THE EXCEPTION TIME_ERROR IS RAISED WHEN THE FUNCTION "-"
--       RETURNS A VALUE NOT IN THE SUBTYPE RANGE DURATION.

-- CPP 8/16/84

WITH CALENDAR;  USE CALENDAR;
WITH REPORT;  USE REPORT;
PROCEDURE C96005D IS

BEGIN
     TEST ("C96005D", "CHECK THAT THE SUBTRACTION OPERATOR RAISES " &
           "TIME_ERROR APPROPRIATELY");

     ---------------------------------------------

     BEGIN     -- (D)

          DECLARE
               NOW, LATER : TIME;
               WAIT : DURATION;
          BEGIN
               NOW := TIME_OF (1984, 8, 13, 0.0);
               LATER := (NOW + DURATION'LAST) + 1.0;
               WAIT := LATER - NOW;
               FAILED ("EXCEPTION NOT RAISED - (D)1");
          EXCEPTION
               WHEN TIME_ERROR =>
                    NULL;
               WHEN OTHERS =>
                    FAILED ("WRONG EXCEPTION RAISED - (D)1");
          END;

          DECLARE
               NOW, LATER : TIME;
               WAIT : DURATION;
          BEGIN
               NOW := TIME_OF (1984, 8, 13, 0.0);
               LATER := (NOW + DURATION'FIRST) - 1.0;
               WAIT := NOW - LATER;
               FAILED ("EXCEPTION NOT RAISED - (D)2");
          EXCEPTION
               WHEN TIME_ERROR =>
                    NULL;
               WHEN OTHERS =>
                    FAILED ("WRONG EXCEPTION RAISED - (D)2");
          END;

     END; -- (D)

     ---------------------------------------------

     RESULT;
END C96005D;
```



Ada Joint Program Office

awards

Ada Validation Certificate # 88060711.09138

to

Telesoft/TeleLOGIC

for successfully Validating

TeleGen2 Sun-386i Ada Compiler Version 3.20

IABG, Federal Republic of Germany
Ada Validation Facility

29 September 1988
Date of Issue

29 September 1989
Expiration date

Tested Configuration

Host(s): Sun-386i (under SunOS, Version 4.0)

Target(s): Same as Host

ACVC Version: 1.9

Das Ada-Validierungszertifikat

Summary Report, VSR) und der Eintrag in die Liste der validierten Compiler. Ein validierungswilliger Compiler-Anbieter wikkelt die Validierung alleine mit der AVF ab. Diese beliefert ihn zunächst mit den aktuellen Testprogrammen, die bereits speziell auf seine Konfiguration zugeschnitten sind. Der Anbieter läßt den gesamten Satz an Testprogrammen bei sich laufen und schickt der Validierungsstelle ein Magnetband mit den Ergebnissen (die sogenannte Prävalidierung). Die Ergebnisse werden dort ausgewertet. Wenn sich dabei kein Grund zur Beanstandung ergibt, dann wiederholt die AVF die gesamte Validierung im Haus des Kunden und vergleicht die Ergebnisse mit denen der Prävalidierung. Wenn auch das erfolgreich ist, dann fertigt die AVF den Ergebnisbericht an und schickt ihn zur Registrierung an die AVO. Der Ergebnisbericht ist öffentlich und gibt sehr genau Auskunft über alle Details der vorliegenden Sprachimplementierung. Das Validierungszertifikat ist für einen gewissen Zeitabschnitt gültig, der vom Inkrafttreten der nächsten ACVC-Version abhängt. Der Modus des Versionenwechsels und der Gültigkeitsdauer der Testprogramme hat sich in der Vergangenheit mehrfach geändert. Zur Zeit beläuft sich die Gültigkeitsdauer auf $1^1/_2$ Jahre (ACVC Version 1.10) bis Ende 1989, wobei ein Compiler noch ein Jahr nach Ende der Gültigkeitsdauer von ACVC 1.10 den Status „validiert" behält. Wie schon erwähnt, soll die ACVC-Suite

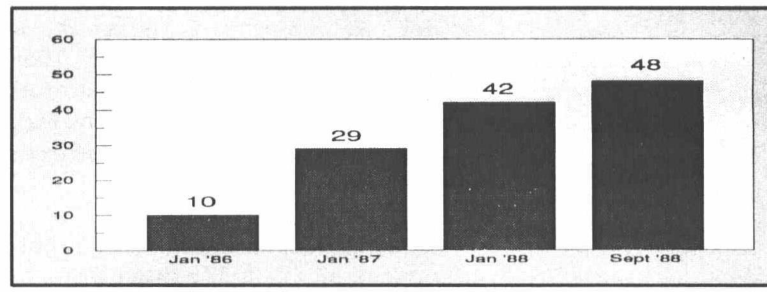
eines Tages eingefroren werden, was auch wieder einen veränderten Validierungsmodus zur Folge haben wird.

Validierungsverfahren sichert Überleben

Wie sieht nun die Fachwelt das Thema „Einsatz von Ada-Compilern"? Bei dem Umstieg auf Ada gibt es immer noch verbreitete Vorbehalte, die bis zum Jahr 1986 sicherlich berechtigt waren. Den damaligen Compilern konnte eigentlich keine Produktionsqualität zugesprochen werden. Dies hat sich mittlerweile gründlich geändert, da die meisten Compiler-Hersteller ihre zweite Generation von Ada-Compilern auf den Markt brachten. Mittlerweile sind für alle gängigen Rechnertypen Ada-Compiler verfügbar. Die Liste der validierten Compiler vom Dezember 1988 verzeichnet 158 validierte Basis-Compiler. Weitere 71 Compiler sind als abgeleitete Compiler eingetragen. Ein abgeleiteter Compiler geht durch unwesentliche Modifikationen aus einem Basis-Compiler hervor und wird nicht der vollen Validierungsprozedur unterworfen. Das Validierungsverfahren ist etabliert und akzeptiert. Ein Compiler, der nicht validiert ist, hat keine Marktchancen. Mehrere staatliche Organisationen (NASA, NATO) und US-Firmen haben Ada als Standard-Programmiersprache gewählt. In Deutschland stehen wir zur Zeit vor einer ähnlichen Entscheidung. In jedem Falle wäre damit die Auflage verbunden, in Projekten nur validierte Compiler einzusetzen.

Ada ist eine mächtige Sprache, die speziell die Entwicklung von umfangreichen Realzeit-Systemen unterstützt.

Jährliche Anzahl der Unternehmen, die validierte Compiler anbieten



Würde die Sprache in Dialekten und Teilmengen zerfließen, dann hätte sie langfristig keine Überlebenschancen. Das Validierungsverfahren ist daher aus heutiger Sicht unerläßlich.

Ada als Software-Engineering-Methode

Dr. Peter Hruschka



Der Autor Dr. Peter Hruschka, 36, ist International Program Manager für CASE-Technologie bei der GEI in Aachen. In dieser Funktion ist er verantwortlich für die langfristige, strategische Planung von Promod, der CASE-Umgebung der GEI.

„Die Entwurfsmethoden von heute sind die Programmiersprachen von morgen." Diese Evolution läßt sich in der Geschichte der Entwurfsmethoden und Programmiersprachen mehrfach belegen. Die Programmiersprache Ada ist dafür ein ganz ausgezeichnetes Beispiel. Allgemeine Software-Engineering-Prinzipien, die in den 70er und anfangs der 80er Jahre bekannt und auch allgemein anerkannt waren, wurden in Ada als Sprachkonzepte eingearbeitet.

Dieser Artikel erläutert einige Entwurfsprinzipien, die in Ada durch Sprachkonstrukte unterstützt werden; er zeigt aber auch, wo die Grenzen der Programmiersprache sind und wo andere Software-Engineering-Mittel eingesetzt werden müssen.

Software Engineering

Seit der NATO-Konferenz 1969 in Garmisch-Partenkirchen, die das Wort „Software Engineering" prägte, haben sich viele Forscher in Universitäten und Industrie mit den Problemen der Software-Krise intensiv auseinandergesetzt und verschiedene Lösungsvorschläge erarbeitet. Dazu zählen die Konzepte der strukturierten Programmierung (von Dijkstra und Wirth), des modularen Entwurfs (von Parnas) und der strukturierten Analyse (von DeMarco). All diese Methoden basieren auf ähnlichen Grundkonzeptionen, die in späterer Zeit auch in Programmiersprachen wie Ada und Modula2 sowie in objektorientierte Sprachen wie C + +, Objective C und andere Sprachen dieser Art übernommen worden sind.

Software Engineering: Ziele und Prinzipien

Den Software-Engineering-Zielen – unter anderem leichte Änderbarkeit, Zuverlässigkeit, Effizienz, Portabilität und Verständlichkeit – kommt man durch konsequente Anwendung einiger Grundprinzipien näher, die auch von Programmiersprachen mehr oder weniger gut unterstützt werden.

Abstraktion

Eines der wichtigsten Software-Engineering-Prinzipien ist das der Abstraktionsebenen oder der schrittweisen Verfeinerung. Unter Abstraktion versteht man die Konzentration auf einige wichtige Eigenschaften und das (vorläufige) Unterdrücken unwichtiger Details einer Komponente. Abstraktion beruht auf der Annahme, daß man eine Komponente auch in Teilkomponenten zerlegen kann. Dieses Prinzip ist zum Beispiel in der Methode der strukturierten Analyse besonders ausgeprägt: man beginnt die Systemdarstellung auf sehr abstrakter Ebene mit einem Überblicksdiagramm (dem Kontextdiagramm) und zerlegt es dann in eine Hierarchie von Datenfluß-Diagrammen, die immer mehr Details enthalten, bis man auf einer Ebene von Diagrammen anlangt, die nur mehr primitive Prozesse oder Funktionen aufweisen.

Herkömmliche Programmiersprachen stellen als Abstraktionsmittel meist nur Unterprogramme (Prozeduren oder Funktionen) zur Verfügung. Ada wurde als eine der ersten Sprachen explizit so entworfen, daß das Prinzip der Abstraktionsebenen in vielfältiger Weise unterstützt wird, zum Beispiel durch Packages (Sammlungen von Funktionen und Daten) und Tasks (parallel ablauffähige Programmeinheiten), durch die Auftrennung von Spezifikationen und Body, durch generische Programmeinheiten (gemeinsame Algorithmen, die mit verschiedenen Datentypen parametriert werden können), durch benutzerdefinierbare neue Datentypen und vieles anderes mehr.

Das Abstrahieren während des Entwurfsprozesses wird im Gegensatz zu anderen Programmiersprachen (wie etwa Pascal) dadurch unterstützt, daß es in Ada genügt, die Spezifikation einer Programmeinheit hinzuschreiben, um sie in anderen Teilen benutzen zu können. Der Body kann später (als getrennte Kompilationseinheit) dazugefügt werden.

Im folgenden Beispiel ist eine mehrstufige Verfeinerung skizziert: Das Paket LACKABFUELLANLAGE enthält ein weiteres Paket FUELLDATEN (das im folgenden

noch verfeinert wird) und zwei parallel ablauffähige Tasks, STEUERE ABFUELLUNG und BERECHNE NETTOGEWICHT. Erstere davon enthält wiederum zwei prozedurartige Programmteile (bei Tasks „Entries" genannt).

```
package LACKABFUELLANLAGE is
  task STEUERE ABFUELLUNG is
  entry LIES TARAGEWICHT (...);
  entry BERECHNE NETTOGEWICHT (...);
  end STEUERE ABFUELLUNG;
  task BEDIENERDIALOG is ...
    :
  end BEDIENERDIALOG;
  package FUELLDATEN is ...
    :
  end FUELLDATEN;
end LACKABFUELLANLAGE;
```

Auf diese Weise kann ein hierarchischer Entwurf unmittelbar nach Ada umgesetzt werden. Wenn man das Problem dann genauer durchleuchtet hat, kann man die algorithmischen Teile getrennt hinzufügen:

```
task body BEDIENERDIALOG is
  ...
end BEDIENERDIALOG;
```

Information Hiding

Ein weiteres Prinzip ist das des Information Hiding. Das Prinzip besagt, daß alles, was außerhalb eines Programmteils nicht sichtbar sein muß, auch verborgen werden sollte. Durch Verbergen von Details, die für die Benutzung eines Programmteils nicht unbedingt notwendig sind, wird die Wiederverwendbarkeit dieses Programmteils erheblich vergrößert. Verborgen werden vor allem Design-Entscheidungen, wie Algorithmen oder wichtige Datenstrukturen.

Das Prinzip des Information Hiding wird in Ada vor allem durch die Trennung der Spezifikation vom Body erreicht. Der Programmierer stellt nach außen nur die Spezifikation zur Verfügung, nach innen (im Body) werden die Algorithmen verborgen; Datenstrukturen und Datentypen lassen sich in Ada am besten durch private Datentypen oder eingeschränkte private Datentypen vor ungewolltem Zugriff schützen.

Lokalitätsprinzip

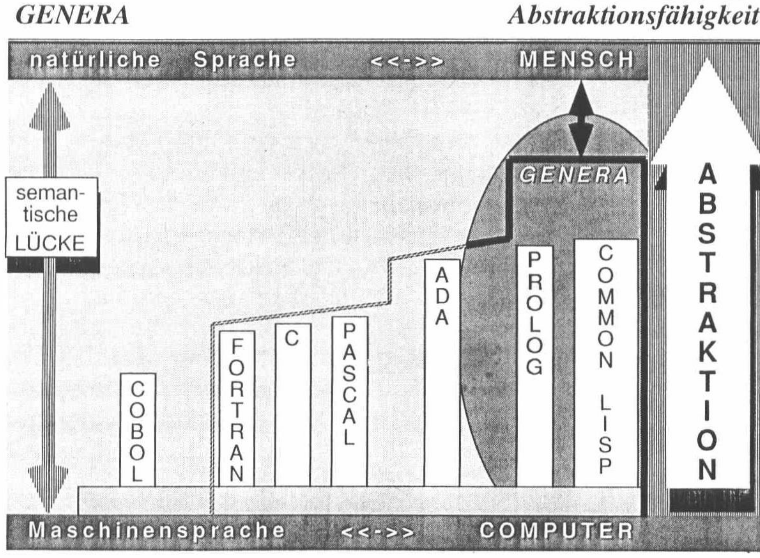
Das Prinzip der Lokalität besagt, daß Objekte in einem Modul möglichst starken Zusammenhang aufweisen sollen. Dadurch werden einzelne Module möglichst klein, logisch stark zusammenhängend und leicht änderbar. Dies wird im strukturierten Entwurf oft auch als „Strength" oder „Strong cohesion" bezeichnet. Auch hier hilft Ada durch die privaten Datentypen dabei, solche „starken" Module zu erstellen. Für die privaten Datentypen werden zwar die Struktur und der Wertebereich vollständig festgelegt, aber diese sind dem Benutzer nicht bekannt. Der Benutzer sieht nur die auf diesem Datentyp erlaubten Operationen. Dadurch wird die Verantwortung für Zugriff auf Variable dieses Typs ganz klar einem Modul zugeordnet. Derartige Module, die alle Zugriffsfunktionen auf einen bestimmten Datentyp zusammenfassen, werden oft auch als abstrakte Datentypen bezeichnet.

Verfeinert man das obige Beispiel weiter, so kann man zum Beispiel das Paket FUELLDATEN als derartigen abstrakten Datentyp in Ada ausformulieren.

```
package FUELLDATEN is
  type ABFUELLDATEN is private;
  type STATUS is (OK, NOT-OK);
  procedure ERFASSE-FUELLDATEN ...
  procedure DRUCKE-FUELLDATEN ...
  function PRUEFE-STATUS (...) return
  STATUS-TYPE;
  private
  type ABFUELLDATEN is
  record ...
  end record;
end FUELLDATEN;
```

Ada – „Bindeglied" zwischen den Programmiersprachen der vierten und fünften Generation

GENERA *Abstraktionsfähigkeit*



Quelle: Symbolics

Literatur: G. Booch: Software Engineering with Ada Benjamin/ Cummings, 1987
R. J. A. Buhr: System Design with Ada Prentice Hall, 1984
T. DeMarco: System Analysis and Specification Prentice Hall, 1979
E. W. Dijkstra: A Discipline of Computer Programming Prentice Hall, 1976
D. L. Parnas: On the criteria to be used in decomposing systems into modules CACM 15, 1972

Modularität

Eng verbunden mit der Lokalität ist das Prinzip der Modularität. Es ist gewissermaßen die gegenteilige Sicht: Module eines Systems sollen so entworfen werden, daß sie untereinander möglichst wenig Verbindungen aufweisen. Dies wird im strukturierten Entwurf „Loose coupling" genannt. Je loser die Module verbunden sind (oder anders ausgedrückt: je weniger sie von anderen Modulen brauchen oder benutzen), desto leichter lassen sie sich aus einem System herauslösen und in anderen Systemen wiederverwenden. Auch die Folgen einer Fehlerkorrektur werden dadurch örtlich gut begrenzt.

Ada stellt dafür das Konzept der generischen Programmeinheiten zur Verfügung. Beim Entwurf solcher generischen Einheiten versucht man gezielt, bestimmte Abhängigkeiten aus dem Modul zu isolieren. So kann man beispielsweise Sortierprogramme schreiben, ohne vorher wissen zu müssen, welche Elemente sortiert werden sollen (Temperaturen, Punkte im dreidimensionalen Raum etc.), oder welcher Sortieralgorithmus angewandt werden soll (Heapsort, Quicksort etc.).

Die Unterstützung des Lokalitäts- und des Modularitätsprinzips in Ada haben wesentlich dazu beigetragen, daß eine neue Industrie entstanden ist, die Standardkomponenten in Ada erzeugt und anbietet. Diese gut wiederverwendbaren und flexibel einsetzbaren Standardkomponenten werden oft – in Analogie zur parallelen Entwicklung von Hardware-Chips – als Software-ICs bezeichnet.

Prinzip der Einheitlichkeit

Als letztes sei noch das Prinzip der Einheitlichkeit erwähnt. Ada trägt durch seine rigorose Definition zu einer gewissen Standardisierung bei. Durch die reichhaltigen Konstrukte, die die Sprache anbietet, lassen sich viele Probleme in sehr natürlicher Weise ausdrücken, statt sie mühsam in Form von primitiven Anweisungen hinschreiben zu müssen. Dies gilt insbesondere auch im Tasking, wo mit dem Rendezvous-Mechanismus ein häufig gebrauchtes Kommunikationsmittel direkt in der Sprache zur Verfügung gestellt wird. Ein zweites Beispiel ist das Exception Handling. In vielen anderen Sprachen wird der Programmcode durch die explizit eingestreute Behandlung von Ausnahmen unübersichtlich, damit schwer lesbar und wartbar. Ada-Programmierer müssen zwar einmal die dafür vorgesehenen Ada-Konstrukte lernen, von dann ab ist Ausnahmebehandlung in Programmen aber leicht zu schreiben.

Die Einheitlichkeit soll unter anderem auch dadurch gefördert werden, daß es für Ada (bis heute) per Definition keine Teilmengen oder Übermengen der Sprache gibt. Sie ist genormt, jeder Compiler muß volles Ada verstehen, was durch Validierung sichergestellt wird. Die Kehrseite davon: Es läßt sich jedoch nicht testen, daß jeder Programmierer volles Ada versteht. Die Mächtigkeit der Sprache erlaubt für die Lösung fast jedes Problems mehrere alternative Sprachmittel, was den Vorteil des rigorosen Sprachstandards wieder etwas einschränkt. De facto arbeiten die meisten Programmierer wieder mit einer Teilmenge, die sie verstehen und beherrschen, manche davon mit der falschen Teilmenge. Sie finden in Ada auch all die Konstrukte wieder, die sie in ihrer alten Sprache schätzen- und liebengelernt haben, und benutzen ausschließlich diese. Dadurch betrügt man sich natürlich um all die Vorteile der Software-Engineering-Prinzipien.

Software Engineering ist mehr als Ada

Eine Programmiersprache alleine kann aber nie all das zur Verfügung stellen, was Software Engineering insgesamt ausmacht.

In diesem Artikel wurden auch nur einige der Spracheigenschaften von Ada aufgezeigt, denen Software-Engineering-Prinzipien zugrunde liegen. Ada ist darüber hinaus eine sehr mächtige Sprache mit vielen modernen Sprachkonstrukten. Ada bietet aber auch (fast) alles, was aus BASIC, Fortran, Pascal und anderen Sprachen bekannt ist. Man sollte daher die Sprache Ada nicht mit einigen der ihr zugrundeliegenden Prinzipien gleichsetzen. Ada kann sehr hilfreich beim Entwurf großer Systeme sein, da dem Programmierer mehr Ausdrucksmittel zur Verfügung stehen als bisher. In den USA ist jedoch inzwischen auch das Wort ADATRAN sehr weit verbreitet, womit auf den (schlechten) Stil einiger Programmierer hingewiesen wird, die zwar syntaktisch korrekte Ada-Programme schreiben, aber noch immer in Fortran-Konzepten denken. Für Mathematiker könnte man also sagen: Ada ist eine notwendige, aber keineswegs hinreichende Bedingung.

Kriterien

Zu der Menge der hinreichenden Kriterien gehören unter anderem auch eine Ada-Programmierumgebung, die im engeren Sinne einen Compiler, ein Laufzeitsystem, einen Linker und Debugger, eventuell auch einen sprachsensitiven Editor und Prettyprinter umfaßt. Darüber hinaus gehören sicherlich eine CASE-Umgebung mit Analysewerkzeugen und Entwurfswerkzeugen dazu sowie Projekt-Managementwerkzeuge, Software-Metriken, Qualitätssicherungsmaßnahmen und vieles andere mehr.

Zusammenfassung

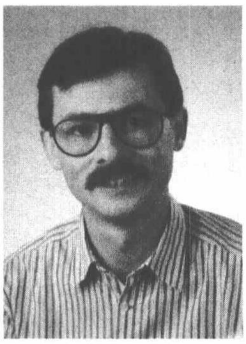
Richtig eingesetzt kann Ada einiges ersetzen, was bis gestern noch nur in Form von speziellen Entwurfsmethoden zur Verfügung stand. Die Sprache enthält mächtige Konstrukte, die Programmsysteme in abstrakte Blöcke aufteilen helfen; Ada unterstützt wie keine andere der weitverbreiteten Sprachen einen Programmierstil, der durch starke Module und schwache Modulkopplung die Wiederverwendbarkeit und Änderbarkeit von Programmen drastisch verbessert. Andererseits ist Ada – wie auch alle anderen Programmiersprachen – natürlich nicht das Allheilmittel für die Software-Krise. Zusammen mit einer guten Software-Engineering-Ausbildung und einer guten Umgebung für Computer Aided Software Engineering wird Ada jedoch dazu beitragen, bessere Programmsysteme zu entwickeln, die auch langfristig gut wartbar sind.

Weitere Informationen

DIN 66268: Programmiersprache Ada. Berlin: Beuth.
Bengel, G.: Ada. Konzepte und Anwendungsbeispiele aus der Prozeßautomatisierung. Düsseldorf: VDI-Verlag, 1986.
Sammet, J.E.: Why Ada is not just another Programming Language. CACM 29 (1986) Nr. 8, S. 722/732.

Der Einsatz von Ada in der Praxis

Hans-Stephan Jansohn und Rolf Holzapfel



Die Autoren:
Dr. Hans-Stephan Jansohn ist vom Jahrgang 1955. 1975–1980 Studium der Informatik. 1980–1984 wissenschaftlicher Mitarbeiter an der Universität Karlsruhe. Promotion 1984. Seit Sommer 1984 Mitarbeiter bei Systeam. Dort unter anderem Entwicklung von Code-Generatoren und Laufzeit-Systemen für Ada.



Rolf Holzapfel
Jahrgang 1959. 1980–1986 Studium der Informatik. Seit Herbst 1986 Mitarbeiter bei Systeam. Dort Mitarbeit an der Entwicklung eines globalen Optimierers für Ada.

Die Firma Systeam beschäftigt sich seit ihrer Gründung im Jahre 1982 mit der Erstellung von Ada-System-Software für unterschiedlichste Rechnerkonfigurationen. 1984 wurde ihr Ada-Compiler für VAX/VMS als erster in Deutschland entwickelter Ada-Compiler validiert. Seitdem ist die Qualität des Ada-Systems, bestehend aus Ada-Compiler und Werkzeugen, laufend verbessert worden, und zwar in punkto größere Funktionalität, bessere Übersetzungszeiten und höhere Code-Qualität. Portierungen auf Siemens/BS2000 und MC68020/Unix wurden durchgeführt und Cross-Compiler für MC68020/OS-9 und MC68020/baremachine entwickelt.

Kernstück des Ada-Systems von Systeam ist der Ada-Compiler, der den vollen Sprachumfang gemäß dem Sprachstandard von 1983 [1] verarbeitet, also auch die optionalen Spracheigenschaften wie Repräsentationsspezifikationen, unterstützt. Dazu kommen Completer, Linker, Debugger und weitere Werkzeuge, die die Entwicklung von Anwendungsprogrammen in Ada unterstützen: ein Werkzeug zum Inspizieren der Programmbibliothek (Lusy), Pretty Printer (Pretty), schneller Syntaxprüfer (Syncheck), Werkzeuge zum Finden nichtinitialisierter Variablen (Noninit), zur Erstellung einer Cross-Referenz-Liste auf der Grundlage semantischer Programminformation (Xref), zur vollständigen Expansion von Namen (Expand), zum Wiedergewinnen der übersetzten Quelle aus der Programmbibliothek (Sourcegen) und zur Programmanalyse (Profile). Das Ada-System ist hardwareunabhängig konzipiert. Dementsprechend ähneln sich die Benutzerschnittstellen der verschiedenen Konfigurationen des Systems sehr: Wer mit dem System auf einer VAX gearbeitet hat, kann es auf einem Siemens-Rechner ohne große Einarbeitungszeit benutzen.

Der Ada-Compiler

Eine Stärke des Systems ist die Unterstützung zweier Sprachmittel, die von anderen Ada-Compilern meist nur unzureichend unterstützt werden: generische Einheiten und Fixpunktarithmetik. Mit diesem Compiler können jederzeit die Rümpfe generischer Einheiten nachübersetzt werden, ohne daß anschließend die Übersetzungseinheiten, die Instantiierungen der generischen Einheiten enthalten, nachübersetzt werden müßten. Ferner unterstützt der Compiler eine unbegrenzte Menge von Fixpunkttypen: Repräsentationsspezifikationen für beliebige Werte des vordefinierten Attributs „small" werden akzeptiert. Damit kann ohne Genauigkeitsverluste zum Beispiel in Dezimalbruchteilen gerechnet werden, was bei der Programmierung von Problemstellungen in kaufmännischen Anwendungen unumgänglich ist, wie folgendes Beispiel zeigt:

```
TYPE dm IS DELTA 0.01;
FOR dm'small USE 0.01;
netto: dm;
mwst: dm;
preis: dm;
...
mwst: = (netto * 14) / 100;
preis: = netto + mwst;
```

Einfaches Retargeting

Der Compiler ist so strukturiert, daß eine Portierung auf einen anderen Gastrechner (genannt Rehosting) oder der Anschluß eines Codegenerators für einen anderen Zielrechner (genannt Retargeting) mit geringem Aufwand möglich ist, aber trotzdem eine sehr gute Codequalität erreicht wird. Die Analysephase faßt die lexikalische, syntaktische und semantische Analyse zusammen. Dabei wird das Ada-Programm in die höhere Zwischensprache DIANA transformiert. DIANA steht für *Descriptive Intermediate Attributed Notation for Ada.* Diese Zwischensprache gilt heute als De-facto-Standard. Sie ist in [2] definiert.

Ausgehend von DIANA findet der eigentliche Übersetzungsprozeß statt. Dieser ist nochmals unterteilt, um mit der nie-

deren Zwischensprache AIM eine ideale Ausgangsbasis für Retargetings zu schaffen. AIM ist eine firmeneigene Entwicklung und steht für *Abstract Intermediate Machine*.

Die Einzelschritte

Im AIM-Generator findet der zielmaschinenunabhängige Teil des Übersetzungsprozesses statt, während der Codegenerator den maschinenabhängigen Teil der Übersetzung durchführt. Im AIM-Optimierer sind Optimierungsverfahren zusammengefaßt, die Datenflußanalyse erfordern. Der AIM-Optimierer kann per Übersetzeroption abgeschaltet werden. Für ein Retargeting muß der Codegenerator ausgetauscht werden; alle anderen Übersetzerteile werden im wesentlichen unverändert übernommen.

Optimierungen

Um eine möglichst hohe Codequalität zu erreichen, werden Optimierungen auf den verschiedensten Ebenen im Übersetzer durchgeführt. Dabei gilt der Grundsatz, daß eine Optimierung so früh wie möglich im Übersetzungsprozeß stattfinden soll. Auf diese Art und Weise wird weniger Zwischencode erzeugt und die Übersetzung beschleunigt. Die in der Literatur veröffentlichten Optimierungsverfahren wurden sorgfältig auf ihre Verwendbarkeit für Ada untersucht. Nur die Verfahren, die einerseits keine Probleme mit der Korrektheit ihrer Anwendung auf Ada-Programme bereiten und andererseits auch eine effektive Codeverbesserung ergeben, wurden in den Übersetzer aufgenommen. Die Verfahren, die Datenflußanalyse erfordern, sind im AIM-Optimierer zusammengefaßt. Im einzelnen sind dort enthalten: Elimination nicht erreichbaren Codes, Elimination toten Codes, Wertpropagation, Konstantenfaltung und algebraische Vereinfachungen, Elimination gemeinsamer Teilausdrücke, Halten von Variablen in Registern, Elimination von Laufzeitprüfungen für Elaborations-, Diskriminanten- und Zugriffsprüfungen.

Der Codegenerator

Auch im Codegenerator wird ein hoher Aufwand investiert, um bestmögliche Codequalität zu erreichen: Unter den verschiedenen möglichen Sequenzen für ein AIM-Konstrukt wird stets die mit der kleinsten Ausführungszeit ausgewählt. So werden Spezialbefehle genau dann eingesetzt, wenn es vorteilhaft ist. Dies hat sich besonders im Codegenerator für den Motorola-Prozessor MC68020 bewährt, wo der „full format addressing mode" nur erzeugt wird, wenn es keine günstigere Alternative gibt.

Übersetzerstruktur

```
          Ada
           ↓
+-----------------+
|    Analyse      |
+-----------------+
           |
         DIANA
           ↓
+-----------------+
|  AIM-Generator  |
+-----------------+
           |
          AIM
           |
           +-------------+
           ↓             |
+-----------------+      |
| AIM-Optimierer  |      ↓
+-----------------+      |
           |             |
           +-------------+
           |
          AIM
           ↓
+-----------------+
|  Codegenerator  |
+-----------------+
           ↓
       Objektcode
```

Gute Codequalität

Die erreichte Codequalität läßt sich am besten durch einige Zahlen demonstrieren. Sie wurden auf einer VAX 8530 von Digital Equipment unter VMS 4.7 gemessen: 5 465 Dhrystones/sec, 4 µsec für einen Prozeduraufruf, 71 µsec für ein Signalrendezvous (das heißt der Rumpf der accept-Anweisung ist leer), 98 µsec für ein vollständiges Rendezvous. Dabei beträgt die Übersetzungsgeschwindigkeit ca. 3 000 Zeilen/min.

Das Ada-Laufzeitsystem und seine Optimierung

Das Laufzeitsystem ist optimal auf den Übersetzer abgestimmt. Es stellt Operationen bereit zur Verwaltung von Speicherhalden, zur Verwaltung und Umschaltung von Ada-Tasks, für die Ausnahmebehand-

lung und für besonders komplexe Fixpunktoperationen. Die Anbindung von Interrupts wird, wie in [1] definiert, mittels Adreßspezifikationen für Task-Entries realisiert. Das Laufzeitsystem wird beim Binden automatisch konfiguriert, so daß nur die wirklich benötigten Operationen hinzugebunden werden. Die Task-Verwaltung stellt 16 Prioritätsstufen und preemptive Scheduling zur Verfügung. Nach Ablauf einer delay-Anweisung nimmt eine Task sofort wieder am Scheduling teil. Die Genauigkeit einer delay-Anweisung hängt vom Zielsystem ab, ist aber in der Regel besser als 0,01 Sekunden.

Der Completer

Der Completer erzeugt Code für die Instantiierungen von generischen Übersetzungseinheiten. Er wird implizit beim Binden aufgerufen, kann aber auch explizit aufgerufen werden.

Der Linker

Der Ada-Linker bestimmt, welche Übersetzungseinheiten zu dem Ada-Programm gehören, legt die Elaborations-Reihenfolge fest und bindet schließlich das Ada-Programm. Mit Hilfe des Werkzeugs Lusy kann man sich die Elaborations-Reihenfolge auflisten lassen. Per Option kann selektives Binden durchgeführt werden. Dann werden nur die Unterprogramme aus einem importierten Paket hinzugebunden, die auch tatsächlich benutzt werden. Dies führt unter Umständen zu einer drastischen Reduzierung der Programmgröße, wenn zum Beispiel nur einzelne Unterprogramme aus einem Mathematikpaket benutzt werden.

Lusy: Ausgaben für das Programm.

```
Elaboration order for the units used by the main program:

        PACKAGE IO_EXCEPTIONS  (29)
        PACKAGE TEXT_IO  (43)
        PACKAGE SYSTEM  (5)
        FUNCTION UNCHECKED_CONVERSION  (15) (GEN)
        PACKAGE BINARY_SEARCH  (49) (GEN)
        PACKAGE BODY BINARY_SEARCH  (62) (GEN)
        PACKAGE INT_BINARY_SEARCH  (61)
        PACKAGE BODY INT_BINARY_SEARCH  (64) (IMP)
        PROCEDURE BODY MAIN  (65)


List of units which depend on PACKAGE BINARY_SEARCH  (49) (GEN) :

    (**) PACKAGE INT_BINARY_SEARCH  (61)
    (**) PROCEDURE BODY MAIN  (65)


List of library units which are used by PROCEDURE BODY MAIN  (65) :

        PACKAGE SYSTEM  (5)
        FUNCTION UNCHECKED_CONVERSION  (15) (GEN)
      . PACKAGE IO_EXCEPTIONS  (29)
        PACKAGE TEXT_IO  (43)
        PACKAGE BINARY_SEARCH  (49) (GEN)
        PACKAGE INT_BINARY_SEARCH  (61)
```

Der Debugger

Mit dem Ada-Debugger steht ein mächtiges Werkzeug zur Programmanalyse während der Testphase zur Verfügung. Es hat eine programmierbare Benutzerschnittstelle mit einer Kommandosprache, deren Syntax an Ada angelehnt ist. Damit ist es zum Beispiel möglich, komplexe Aktionen zu Unterprogrammen zusammenzufassen, die an Haltepunkten ausgeführt werden können. Auch ist es möglich, jederzeit die Programmausführung zu unterbrechen und die Kontrolle an den Debugger zu übergeben. Somit bildet der Debugger auch die Ausgangsbasis zur Realisierung individuell gestaltbarer Testwerkzeuge. Der Debugger ist speziell auf die Programmiersprache Ada zugeschnitten; so ist die Ausgabe von strukturierten Objekten in Aggregatform ebenso selbstverständlich wie der Zugriff auf die verschiedenen Tasks und die Unterstützung bei der Ausnahmebehandlung.

Programmbibliotheken

Die Übersetzung einer Ada-Übersetzungseinheit geschieht immer im Kontext einer Programmbibliothek, die man als eine Menge bereits übersetzter Einheiten ansehen kann. Die anfängliche Programmbibliothek enthält die *vordefinierten Übersetzungseinheiten*, das sind: **system, calendar, unchecked conversion, unchecked deallocation, io exceptions, text io, sequential io, direct io und low level io**.

Durch das erstmalige erfolgreiche Übersetzen einer Einheit wird diese Einheit der Programmbibliothek hinzugefügt. Danach kann sie von anderen Übersetzungseinheiten benutzt, zum Beispiel durch eine with-Klausel importiert werden. Beim Nachübersetzen einer Übersetzungseinheit werden alle Übersetzungseinheiten in der Programmbibliothek ungültig, die von der nachübersetzten in irgendeiner Form abhängen. Sie müssen

dann wieder übersetzt werden, damit sichergestellt ist, daß eventuelle Änderungen in der nachübersetzten Einheit in ihren Auswirkungen auch in allen abhängigen Einheiten berücksichtigt werden.

Es sind eine Reihe verschiedener Abhängigkeiten zwischen Übersetzungseinheiten möglich:

- Der Rumpf eines Bibliothekspakets oder -unterprogramms hängt von seiner Spezifikation ab,
- eine Übersetzungseinheit hängt von allen importierten (das heißt in einer with-Klausel aufgeführten) Einheiten ab,
- eine Untereinheit hängt von ihrer Vatereinheit ab oder
- eine Übersetzungseinheit, in der ein Unterprogramm offen eingebaut wurde, weil Pragma inline für dieses Unterprogramm spezifiziert ist, hängt von der Einheit ab, die den Rumpf des Unterprogrammes enthält.

Durch das Werkzeug Lusy kann sich der Programmierer jederzeit ein Bild vom Zustand seiner Programmbibliothek machen. Neben dem Auflisten der in der Programmbibliothek enthaltenen Übersetzungseinheiten kann er sich die Abhängigkeiten anzeigen lassen.

Folgende Operationen werden zur Behandlung von Programmbibliotheken zur Verfügung gestellt: Createlib, Deletelib, Freeze, Unfreeze, Mergelib, Copylib und Changelibname. Dies ermöglicht hierarchische Programmbibliotheken: Jede Programmbibliothek kann durch die Operation Freeze eingefroren und als Vaterbibliothek einer neuen Bibliothek bei der Operation Createlib angegeben werden. Die neue Bibliothek ist dann eine logische Kopie der Vaterbibliothek, in der normal gearbeitet werden kann. Es findet allerdings kein physikalisches Kopieren statt, so daß Plattenspeicherplatz eingespart wird. Auf diese Weise kann ausgehend von einer Programmbibliothek in einem Projekt Parallelentwicklung stattfinden, oder es lassen sich zu einer Spezifikation verschiedene Versionen des Rumpfes parallel halten. Durch die Operation Mergelib kann eine Programmbibliothek mit ihrer Vaterbibliothek verschmolzen werden.

Validierung

Die Validierung eines Ada-Compilers findet durch eine unabhängige, offizielle Einrichtung, das Validierungsbüro (AVO – Ada Validation Office) statt. Anhand einer großen Menge von Testprogrammen, der ACVC-Testsuite (ACVC – Ada Compiler Validation Capability), die zur Zeit cirka 4 000 Tests umfaßt, wird dabei geprüft, ob der Compiler den Sprachstandard von Ada einhält, wie er in [1] definiert ist.

Qualitätssicherung

Neben der Validierung der Systeam Ada-Compiler, die die erfolgreiche Bearbeitung der rund 4 000 Testprogramme umfaßt, werden für jede Version des Ada-Systems eine Reihe weiterer Tests gefahren. Kein Compiler und kein Werkzeug wird ausgeliefert, bevor es nicht jeden dieser Tests erfolgreich bestanden hat. Dazu gehören sämtliche Tests der PIWG-Testsuite, deren Ziel es ist, die Codequalität und die Übersetzungsgeschwindigkeiten zu ermitteln. PIWG ist eine Abkürzung für eine Arbeitsgruppe von SIGAda, nämlich für *Performance Issues Working Group*. Diese Gruppe verbreitet weltweit Testprogramme zur Beurteilung von Ada-Compilern. Außerdem zählen das Vandenberg-Testprogramm [3] sowie Tests zum Kapitel 13 von [1] dazu. Die Kapitel-13-Tests testen Repräsentationsspezifikationen, die von allen Systeam-Ada-Compilern vollständig unterstützt werden. Meßprogramme, besondere Tasking-Tests und versionsspezifische Tests setzen wir ebenfalls ein. Aber nicht nur der Compiler, sondern auch sämtliche Werkzeuge werden eigenen Prüfungen unterzogen.

Das Ada-System ist selbst in Ada implementiert. Dies gestattet es, bei der Entwicklung des Ada-Systems das eigene Ada-System zu benutzen. Dadurch ist es dauernd im Einsatz. Immerhin umfaßt das System etwa eine halbe Million Quellzeilen. Die Erfahrungen zeigen, daß dessen Pflege und Verwaltung mit dem Ada-System problemlos möglich sind.

Die erfolgreiche Selbstübersetzung des Compilers (das sogenannte Bootstrap), das erfolgreiche Bestehen aller Validierungs- und sonstigen zusätzlichen Tests sowie der einwandfreie Dauereinsatz in der täglichen Praxis weisen auf einen hohen Standard hin. Die Codequalität des Compilers reicht aber keineswegs nur für die Programmierung von Compilern aus, sondern genügt auch den hohen Anforderungen, die Realzeit-Anwendungen stellen. Dadurch ist der Weg zu dem Anwendungsgebiet geebnet, für das Ada entworfen wurde.

Literatur

[1] The Programming Language Ada Reference Manual, American National Standards Institute, Inc., ANSI/MILSTD-1815A-1983, erschienen als Band 155 der Lecture Notes in Computer Science, Springer-Verlag, 1983

[2] DIANA. An Intermediate Language for Ada, Revised Version, erschienen als Band 161 der Lecture Notes in Computer Science, Springer Verlag, 1983

[3] Ada Case 01 Study, Issue 1, European Space Agency, Directorate of Operations, Engineering Department 1986

Was spricht für den Einsatz von Ada?

Stefan Klusmeier



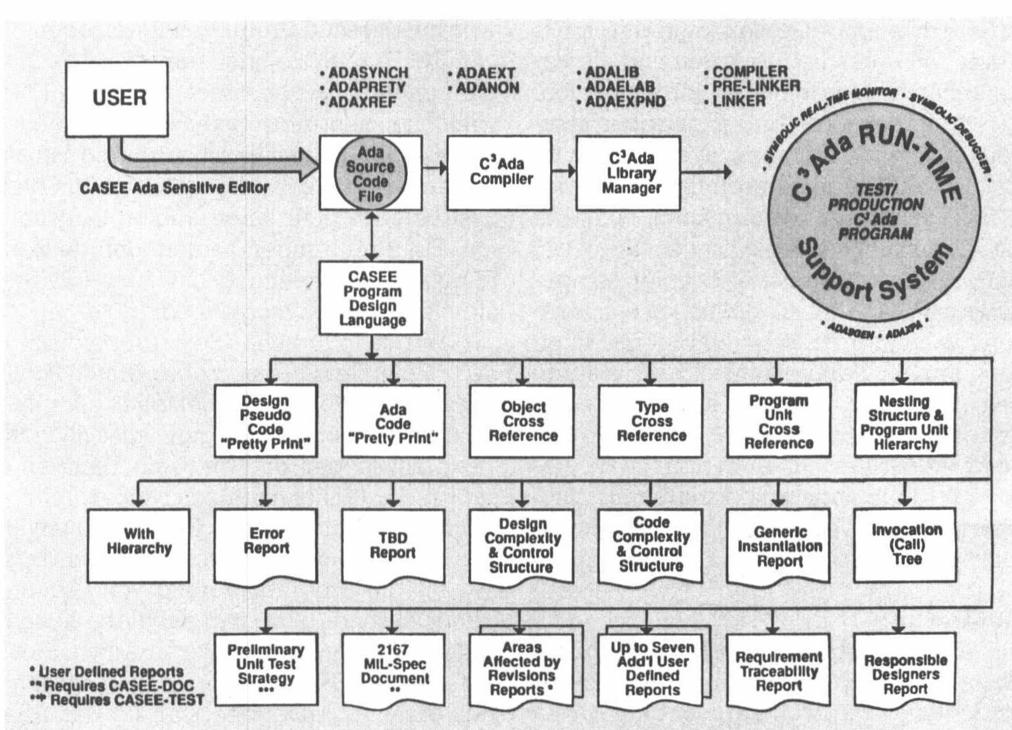
Der Autor: Dipl.-Math. Stefan Klusmeier, 43, Marketing-Leiter bei der Concurrent Computer GmbH, Planegg/Martinsried.

Weiter steigender Bedarf an Software-Applikationen, Anwendungsrückstau, zunehmender Wartungs- und Pflegeaufwand sowie Mangel an qualifizierten Fachleuten waren und sind die Ursachen der in den vergangenen Jahren so oft zitierten „Software-Krise“. Vor dem Hintergrund dieser Entwicklungen formulierte das Department of Defense in den siebziger Jahren seinen Wunsch nach effizienteren Entwicklungsumgebungen, leichter wartbaren Programmen und einem Sprachstandard insbesondere für Anwendungen bei „Embedded Systems“, die weit über 50 Prozent der Anwendungen des DoD ausmachten.

In den 70er Jahren wurde eine Vielzahl von Programmiersprachen für Embedded Systems verwendet. Man hoffte, mit einer einzigen Sprache sowohl ein breiteres Potential kompetenter Programmierer als auch die Möglichkeit zu schaffen, einmal entwickelte Software-Module in anderen, vergleichbaren Projekten erneut einsetzen zu können (Reuseability). Bei der Analyse der vorhandenen Programmiersprachen wurde festgestellt, daß keine existierende Sprache den Anforderungen nach Wiederverwendbarkeit von Software-Teilen, weitgehender Implementierung von akzeptierten Software-Engineering-Grundsätzen und damit hoher Produktivität sowie der gleichzeitig notwendigen Effizienz insbesondere für „Embedded Systems“ erfüllte. Dabei ist vorrangiges Ziel des Software-Engineering das Bemühen, zuverlässige Software schneller zu entwickeln. Sowohl Fortran als auch COBOL wurden wegen ihrer geringen Eignung für die gestellten Aufgaben aus der weiteren Betrachtung ausgeschlossen. Pascal hingegen ging wegen geeigneter Elemente in die Planungen der neuen Sprache ein.

Ein weiteres wesentliches Moment für Ada, das keine andere Programmiersprache vorher voll erfüllte, sind die speziellen Anforderungen der sehr großen Software-Projekte mit inzwischen über elf Millionen Statements (geschätzt für Space Station), die insbesondere im wehrtechni-

Das C³ Ada-Sprachsystem



schen und im Luftfahrt- und Raumfahrtbereich anstehen. Große Projekte können nicht mehr von kleinen Teams realisiert werden. Die Zusammensetzung des Entwicklungsteams wechselt zwangsläufig bei Entwicklungszeiten von mehreren Jahren. Die Wartung der entwickelten Software, die oft für eine Lebensdauer von 15 Jahren geplant ist, muß von immer wieder wechselnden Programmierern effizient durchführbar sein. Entwicklungsteams mit unterschiedlichen Know-how-Schwerpunkten müssen trotz örtlicher Trennung sehr eng zusammenarbeiten können. Schriftlich sind Projekt-Management, Revisionskontrolle und Dokumentations-Aufgaben einer Größenordnung zu bewältigen, die nicht mehr vom Prozeß der eigentlichen Software-Entwicklung getrennt werden dürfen, um konsistent, korrekt und damit nutzbar zu bleiben.

In der Sprache der Informatiker heißt dies: Forderung nach hoher Modularität ohne Aufgabe strenger automatischer Konsistenzprüfungen, nach Trennung von Datenstrukturen und Programmcode, nach lesbaren, selbstdokumentierenden Programmen sowie nach einer vollständigen Programmierumgebung mit einheitlichen, rechnergestützten, aber herstellerunabhängigen Werkzeugen, mit Software-Modul-Bibliothek und nicht nur einem Sprachübersetzer. Ada wurde für all diese Anforderungen konzipiert.

Software Engineering-Konzepte

Keines der Ada-Konzepte ist völlig neu, jedes war bereits in der einen oder anderen Variante in einer anderen Sprache vorhanden. Aber keine andere Sprache vereinigte alle Konzepte in sich. Die wichtigsten sind:

- Strenge Typvereinbarungen;
- Generics;
- Tasking;
- Datenabstraktion;
- Ausnahmebehandlung (Exception Handling);
- Modularität und getrennte Übersetzung.

Strenge Typvereinbarungen

Eine Typvereinbarung definiert eine Menge von Werten und Operationen auf diesen Werten. Enumeration Types (Aufzähltypen) zum Beispiel erlauben es dem Programmierer, abstrakte Datenkonstrukte wie Schiffsklassen zu bilden – Kreuzer, Schlachtschiff, Flugzeugträger, etc. – ohne Rücksicht auf die interne numerische Repräsentation dieser Typen im Computer. Ada erlaubt die Vereinbarung benutzerspezifischer Datentypen, und es erfordert die Typenvereinbarung für alle im Programm benutzten Objekte, um eine strenge Überwachung der Einhaltung des vereinbarten Typs von der Spezifikation bis zur Laufzeit durchzuführen. Anders als zum Beispiel in C werden Programmierfehler in diesem Bereich vom Ada-Compiler selber erkannt.

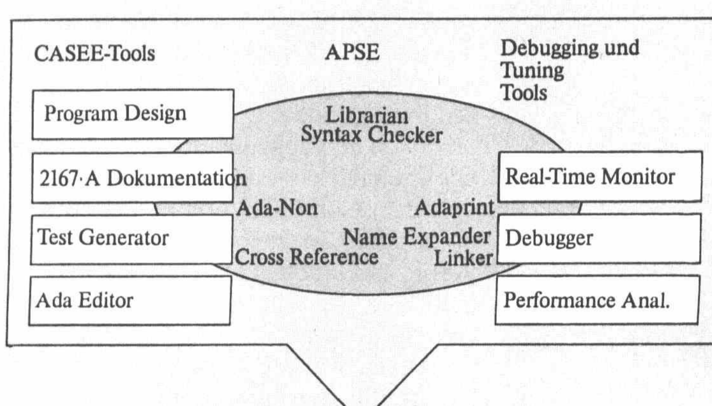
Generics

Generische Software-Module sind Muster für Programm- oder Datenstrukturen, nach denen zur Zeit der Übersetzung entsprechend den eingesetzten Übergabeparametern der konkrete Programmteil vom Computer generiert wird (Instantiation). Damit hat der Programmierer ein sehr leistungsfähiges Werkzeug, um Zusammenhänge allgemein zu beschreiben und die Umsetzung entsprechend den jeweils benutzten Datentypen etc. dem Übersetzer zu überlassen.

Tasking

Mit dem Konzept des Taskings, also gleichzeitig ablauffähiger Programmzweige, ermöglicht Ada parallele Programmierung, die insbesondere bei Echtzeit-Prozessen, zum Beispiel für „Embedded Systems", unverzichtbar ist. Ohne Kenntnis von der unterliegenden Hardware bietet Tasking alle notwendigen Kommunikations- und Synchronisationsmittel.

C³ Ada-Entwicklungswerkzeuge



Datenabstraktion

Datenabstraktion bedeutet die Trennung der Darstellungsdetails der Daten von den abstrakten Operationen, die in ih-

nen definiert wurden. Die Datenabstraktion ermöglicht zusammen mit Generics die Entwicklung von wiederverwendbaren Software-Komponenten.

Ausnahmebehandlung

Eine Ausnahme besteht darin, daß eine Operation in Gang gesetzt wurde, aber nicht auf die übliche Art ausgeführt werden kann. Wenn zum Beispiel eine Operation, die eine Datei lesen soll, mit dem Prädikat „Es existiert noch ein weiterer Datensatz" oder „Unter dem angegebenen Namen existiert eine Datei" versehen ist und eine der Bedingungen nicht erfüllt, besteht eine Ausnahmesituation.

Der Vorteil, den die Ausnahmebehandlung bietet, liegt darin, daß Operationen auf allgemeinere Situationen ausgedehnt werden können. Diese allgemeinere Anwendung beruht darauf, daß die Antwort auf eine Ausnahme nicht von der Anwendung, sondern vom Anwender definiert wird; denn schließlich ist es der Anwender, der die Voraussetzungen definiert und somit die möglichen Ausnahmesituationen übersehen kann. Dadurch kann die Ausnahme angemessener bearbeitet werden.

Modularität und getrennte Kompilierung

Das Package ist der grundlegende Ada-Baustein. Es enthält eine Spezifikation der Möglichkeiten und einen Teil, der sie verwirklicht (Body). Die Spezifikationen können zum Zwecke des Programm-Designs kompiliert werden. Die Ausführungsteile (Bodies) können separat, aber erst nach den Spezifikationen der entsprechenden Packages kompiliert werden, damit der Compiler erkennen kann, ob beide den Spezifikationen entsprechen. Wenn die Spezifikation verändert wird, muß das Ausführungsmodul (Body) rekompiliert werden.

Diese strikte Trennung von Spezifikationen und der korrespondierenden Notwendigkeit der erneuten Kompilierung von abhängigen Programmteilen ist einer von Adas größten Vorzügen hinsichtlich der Zuverlässigkeit der Software. Fehler, die sich durch das Fehlen solcher Test-Routinen des Compilers in die Software einschleichen können, sind äußerst schwierig zu entdecken.

Die Ada-Entwicklungsumgebungen

Entsprechend seiner Ableitung von Pascal versucht Ada, den Programmierer unabhängig von der Hardware arbeiten zu lassen. Dies steht in direktem Gegensatz zu Systemprogrammiersprachen wie C, deren maschinennahe Eigenschaften auf Kosten rigider Typen-Checks oder anderer Konsistenz-Checks verwirklicht werden, wie sie in Ada verfügbar sind. Gerade bei Embedded Systems scheint eine Abwägung zwischen direkter Kontrolle und guter Portierbarkeit notwendig zu sein. Ada legt den Schwerpunkt auf Portierbarkeit und Sicherheit.

Produktivitäts-Tools für Ada

Weil Ada besonders für höchst schwierige Projekte, wie beispielsweise Embedded Systems, gedacht ist, gibt es eine wesentlich größere Nachfrage nach produktivitätssteigernden Programmier-Tools als bei anderen Programmiersprachen. Sogar für C ist das Angebot wesentlich geringer, und die innovativsten Tools kommen aus dem PC-Bereich.

Im Gegensatz dazu bietet Ada von Hilfsmitteln für die einfache Geräteansteuerung über standardisierte computerunterstützte Software-Engineering-Tools („CASE"-Tools) bis hin zu vollständigen Ada-Programmier-Umgebungen (APSE) ein breites Spektrum von Werkzeugen zur Unterstützung effizienter Entwicklung großer Projekte. Heute existieren am Markt etwa 200 Ada-Compiler für die verschiedensten Systeme – vom PC über die Workstation bis hin zum echtzeitfähigen Multiprozessorsystem. Diese Systeme bieten vieles für die effiziente Programmentwicklung.

Impressum

CHIP PLUS erscheint monatlich als Beilage zu CHIP. Anschrift der Redaktion: Schillerstr. 23a, 8000 München 2, Tel. 0 89-5 14 93-0, Verlag: Vogel Verlag und Druck KG, Max-Planck-Str. 7/9, 8700 Würzburg, Tel. 09 31/4 18-0. Redaktionsdirektor: Richard Kerler. Redaktion: Martin Stübs (verantw.), Ulrich Parthier. Grafische Gestaltung: Hans Kuh, Gabi Raffius. Chef vom Dienst: Reiner Lanninger. Schlußredaktion: Dr. Günter Neumann. Autoren dieser Ausgabe: Rolf Holzapfel, Dr. Peter Hruschka, Stefan Klusmeier, Michael Tonndorf, Hans-Stephan Jansohn. Verlagsdirektor: Dr. Andreas Kaiser. Anzeigen: Friedrich Mangold. Satz, Litho, Druck, Verarbeitung, Versand: Vogel Verlag und Druck KG. Der Verlag haftet nicht für unverlangt eingesandte Manuskripte und Fotos.

CHIP WISSEN

Die kompetente Buchreihe rund um den PC



Thomas Geise:

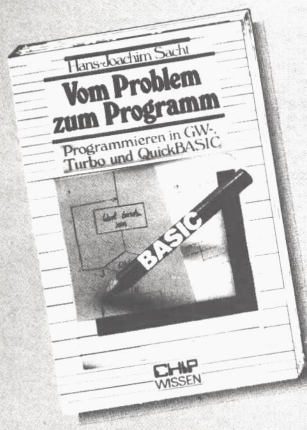
Programmieren mit Turbo BASIC

176 Seiten, 104 Bilder, Hardcover
35,– DM/ISBN 3-8023-**0218**-4

BASIC ist nach wie vor die Programmiersprache Nummer 1. Dieses Buch führt zunächst in die Sprachelemente ein, die sich von einfachen BASIC-Versionen unterscheiden. Das Prozedurkonzept wird ausführlich besprochen. Dem Benutzer werden folgende Unterprogramme zur Verfügung gestellt:

★ Dateiverwaltung
★ Textverarbeitung
★ Grafik

Nebenbei erfährt der Anwender auch viele Tips und Tricks. Das Kapitel Baumstrukturen zeigt moderne Methoden des Zugriffs auf umfangreiche Datenbestände, die zugleich die wichtigsten Anwendungen für Rekursionen sind.
Eine Diskette mit den im Buch behandelten Programmen ist erhältlich.



Hans-Joachim Sacht:

Vom Problem zum Programm

Programmieren in GW-BASIC, Turbo BASIC und QuickBASIC
464 Seiten, 183 Bilder
4., völlig überarbeitete Auflage 1988
48,– DM/ISBN 3-8023-**0214**-1

Das von Microsoft entwickelte GW-BASIC bzw. BASICA gilt inzwischen bei 16-bit-Computern als Standard und für 32-bit-Rechner als Grundlage des Sprachumfangs auch anderer neuer Dialekte.
Die Neukonzeption gliedert sich in 3 Teile:

1. Teil erklärt Grundlagen der BASIC-Programmierung unter Einbeziehung der wichtigsten MS-DOS-Kommandos, Compiler.
2. Teil bringt zahlreiche Programmroutinen für breite Anwendungen.
3. Teil zeigt, wie ein Programm Schritt für Schritt entsteht.

Eine Diskette mit den im Buch behandelten Beispielen ist erhältlich.



Bernhard Ritter:

Produktionsplanung und -steuerung mit Personalcomputern

112 Seiten, 48 Bilder, Hardcover
Preis inkl. Demodiskette: 35,– DM
ISBN 3-8023-**0220**-6

Immer mehr Klein- und Mittelbetriebe sehen sich vor die Frage gestellt: „Ist für unser Unternehmen ein sog. Produktionsplanungs- und Steuerungssystem (PPS-System) sinnvoll und rentabel?“

Dieses Buch zeigt dem Leser in klarer und übersichtlicher Form – unterstützt durch zahlreiche Tabellen und Abbildungen – was er vor, bei und nach der Einführung eines PPS-Systems berücksichtigen muß.
Eine Diskette mit den im Buch vorgestellten Funktionen und Möglichkeiten dieses Systems ist erhältlich.



Bernd Kretschmer:

Multiplan 3.0 auf IBM PC und MS-DOS Rechnern

mit Multiplan Junior

384 Seiten, 268 Bilder
2., völlig überarbeitete Auflage 1988
48,– DM/ISBN 3-8023-**0883**-2

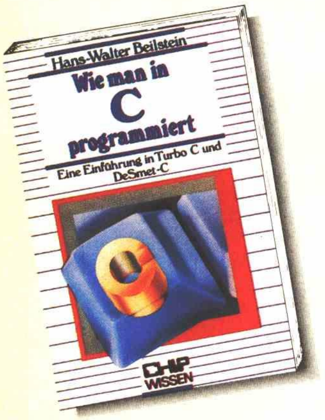
Sie werden feststellen, daß Multiplan 3.0 sehr leicht zu erlernen ist.
Anhand übersichtlicher Beispiele zu Prozentrechnung, Umsatzstatistik, Text- und Textbausteinverarbeitung, Erstellen von Serienbriefen, Dateibearbeitung sowie Lieferscheinbeschreibung lernen Sie, wie Multiplan 3.0 arbeitet.
Alle erforderlichen Eingaben am PC sind genau beschrieben und Bildschirminhalte/-Ausdrucke so abgebildet, wie sie Ihr PC darstellt.

Haben Sie schon den neuen „CHIP- Katalog“?
Bestellen Sie gleich!

VOGEL Buchverlag Würzburg
Postfach 6740 · 8700 Würzburg 1

CHIP WISSEN

Die kompetente Buchreihe rund um den PC



Hans-Walter Beilstein:

Wie man in C programmiert

Eine Einführung anhand von Turbo C und DeSmet-C
504 Seiten, 143 Bilder, Hardcover
53,– DM/ISBN 3-8023-**0189**-7

Dieses Buch führt gründlich und umfassend in die Besonderheiten von C ein. C-Kenntnisse werden nicht vorausgesetzt, es ist aber von Vorteil, wenn der Leser sich schon mit einer anderen Programmiersprache beschäftigt hat. Zunächst erfährt er alles über C (u.a. C-Befehle und C-Funktionen, Ein- und Ausgaberoutinen, Variablen, C-Vektoren, C-Bibliothek), dann werden C-Tools wie Präsentationsgrafik oder Windowtechnik angewendet und in vorhandene Programme eingebaut. Aufgaben mit Lösungen zu jedem Kapitel helfen, die erworbenen Kenntnisse anzuwenden und zu überprüfen.

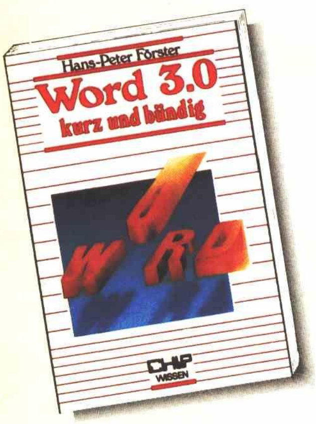


Hans-Peter Förster/Martin Zwernemann:

MultiMate 3.6 kurz und bündig

144 Seiten, 13 Bilder, Hardcover
33,– DM/ISBN 3-8023-**0208**-7

MultiMate ist ein komfortables, leicht bedienbares Textprogramm. Seine Kombination aus Menüsteuerung und Funktionstastenprinzip ermöglicht eine ganz auf das persönliche Vorgehen zugeschnittene Arbeitsweise. Das Buch beschreibt in kompakter Form die Funktionen und den Umgang mit MultiMate. Es richtet sich an interessierte Einsteiger, die schnell den Umgang mit diesem Programm beherrschen wollen. Alle MultiMate-Befehle stehen griffbereit zur Verfügung. Das Buch ist somit eine nützliche Ergänzung zu den Handbüchern.

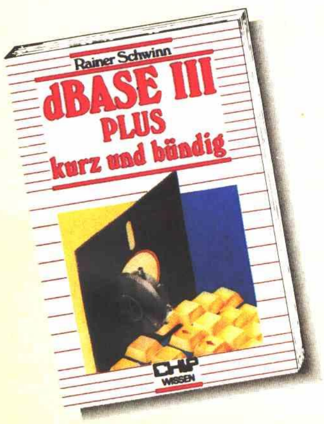


Hans-Peter Förster/Martin Zwernemann:

Word 3.0 kurz und bündig

Texte erfassen, gestalten, drucken
200 Seiten, 61 Bilder, Hardcover
38,– DM/ISBN 3-8023-**0188**-9

Dieses Buch erklärt die Funktionen der deutschen Version 3.0 des Textprogramms MS-Word. Es hilft dem Anwender, alle Word-Befehle auf kompaktem Raum stets griffbereit zu haben. Word 3.0 steuert Laserdrucker und Satzmaschinen direkt an. So lassen sich sehr flexibel Druckwerke produzieren. Zusammen mit einer Datenbank und einer Desktop-Software sind die Anwendungsmöglichkeiten äußerst vielfältig. In kurzer verständlicher Form sprechen die Autoren auch diese Punkte an und stellen anhand praktischer Beispiele interessante Anwendungen vor.



Rainer Schwinn:

dBase III PLUS kurz und bündig

260 Seiten, 26 Bilder, Hardcover
38,– DM/ISBN 3-8023-**0180**-3

dBase III PLUS unterscheidet sich von seinem Vorgänger dBase III durch einen erweiterten Leistungsumfang, z.B. seine Netzwerkfähigkeit. Der Leser wird anhand leicht nachvollziehbarer Beispiele in das Arbeiten mit der deutschen Version dieser Datenbank eingewiesen. Dabei lernt er u.a. wie er

- die Benutzerführung durch einen Assistenten in Anspruch nimmt und das System um Hilfe fragt
- Dateistrukturen erstellt, verändert und kopiert
- Daten in Dateistrukturen eingibt und modifiziert
- individuelle Bildschirmmasken erzeugt
- Dateien sortiert und indiziert
- wie er mit einem Programmgenerator ohne Programmierkenntnisse eigene Programme erstellt.

Haben Sie schon den neuen „CHIP-Katalog“?
Bestellen Sie gleich!

VOGEL Buchverlag Würzburg
Postfach 6740 · 8700 Würzburg 1